COMP 30230 Connectionist Computing
MLP Programming Assignment

Kevin O'Brien
12498432

To complete this assignment, I created an MLP with a single hidden layer written in Python using an OOP programming style. The MLP is constructed in such a way that the user can specify to use Sigmoid, Hyperbolic Tangent, Rectified Linear Unit, or Softmax activation functions in the hidden and output layers. In the case that Softmax is chosen for the output layer, the user specifies which of the other three to use in the hidden layer.

I also created a class named DataSet, which takes into its constructor a 2-dimensional list of inputs and corresponding labels, as well as a fraction to divide the data into training and test sets. This makes handling the data easier, and allowed for helpful MLP.train() and MLP.test() methods to be created, which train and test for one epoch on the entire training and test sets, respectively. Once the MLP is trained, new unseen data can easily be fed into it using the MLP.forward() function, and also retrained if required using MLP.backwards()

XOR

This script can be run using the command

```
python XOR.py <activation function> [<hidden activation function>]
```

where <activation function> is specified as one of the following:

- sigmoid
- tanh
- relu
- softmax

and <hidden activation function> must be specified when using softmax output activation.

e.g. `python XOR.py sigmoid`

`python XOR.py softmax relu`

The labels are set as [1, 0] = 0 and [0, 1] = 1 when using softmax at the output.

The MLP is trained for 10,000 epochs of the four examples in the data set, using the best learning rate for the given activation functions found through my experimenting. All activation functions were able to easily achieve 100% accuracy very quickly, but there were quite substantial differences in the minimum loss values that were reached at the end of training.

Sigmoid activations reached a loss value as a factor of e^{-5}, tanh reached e^{-9}, and ReLU was far better reaching e^{-30} (however, ReLU often got stuck in a local minimum with a loss value plateauing at ~0.252 and 0% accuracy, and had to be re-ran). Softmax performed the worst reaching a minimum loss of ~0.0002 with ReLU hidden activations, however using softmax here is pointless anyway as it is just a binary classification problem. Interestingly, softmax followed the same trend depending on the hidden activations, with sigmoid being the worst, then tanh, and ReLU being the best.

Here is the training output when using ReLU:

```
EPOCH:       0
TRAIN LOSS:  0.5961961001004048
TRAIN ACC:   75.00 %

EPOCH:       1000
TRAIN LOSS:  3.5097826581212955e-09
TRAIN ACC:   100.00 %

EPOCH:       2000
TRAIN LOSS:  1.3865907988348844e-19
TRAIN ACC:   100.00 %

EPOCH:       3000
TRAIN LOSS:  1.5248067856483038e-29
TRAIN ACC:   100.00 %

EPOCH:       4000
TRAIN LOSS:  5.451728152839408e-30
TRAIN ACC:   100.00 %

EPOCH:       5000
TRAIN LOSS:  5.447776057961296e-30
TRAIN ACC:   100.00 %

EPOCH:       6000
TRAIN LOSS:  5.445635863586688e-30
TRAIN ACC:   100.00 %

EPOCH:       7000
TRAIN LOSS:  5.443160774968744e-30
TRAIN ACC:   100.00 %

EPOCH:       8000
TRAIN LOSS:  5.444237925176209e-30
TRAIN ACC:   100.00 %

EPOCH:       9000
TRAIN LOSS:  5.442954822981523e-30
TRAIN ACC:   100.00 %
```

$\sin(x_1 - x_2 + x_3 - x_4)$

This script can be run as

```
python Sin.py
```

where the parameters (number of hidden neurons, learning rate, training epochs, etc.) can be set at the top of the script.

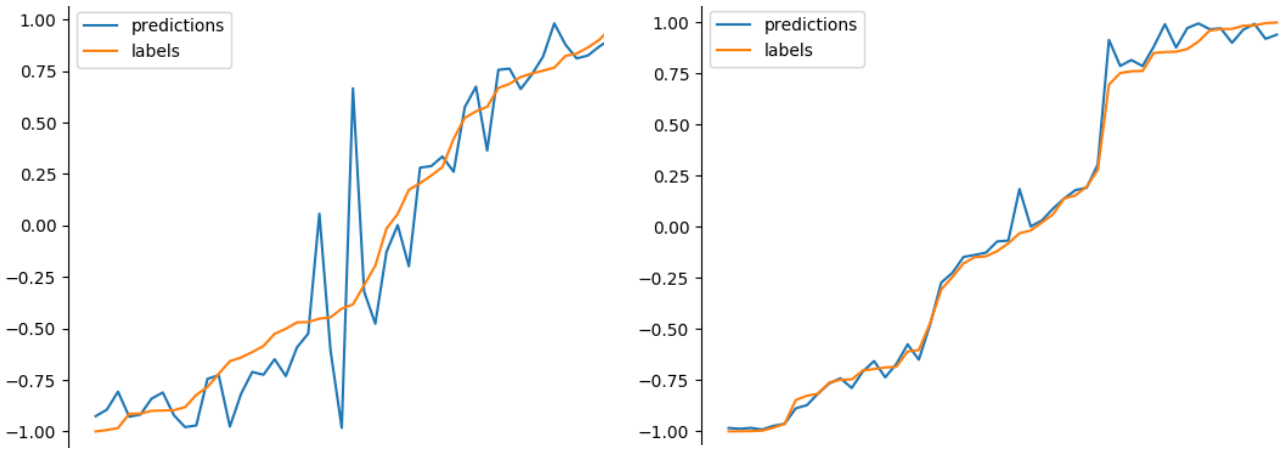
This was an interesting task. I used my MLP with tanh activations here, as its output range matches that of the sin function. Using the minimum specifications given in the assignment (5 hidden units and 50 total data points), I saw substantially varying results. Somethimes, the training and testing loss values would both decrease consistently together, other times the test loss would begin to increase, presumably as the MLP began to overfit the small data set.

The best results I achieved after 10,000 training epochs with these minimum specifications were as follows:

```
EPOCH:       0
TRAIN LOSS:  0.8524717490390504
TEST LOSS:   0.8785463780326553

EPOCH:       1000
TRAIN LOSS:  0.008704028900173754
TEST LOSS:   0.007271947093044097

EPOCH:       2000
TRAIN LOSS:  0.006578697074696394
TEST LOSS:   0.007630047349498887

EPOCH:       3000
TRAIN LOSS:  0.00441529612828806
TEST LOSS:   0.004331973383689467

EPOCH:       4000
TRAIN LOSS:  0.003604501841496875
TEST LOSS:   0.002368302072122374

EPOCH:       5000
TRAIN LOSS:  0.0031651165405203593
TEST LOSS:   0.0013974351606650466

EPOCH:       6000
TRAIN LOSS:  0.002902233395483066
TEST LOSS:   0.00105398168089686

EPOCH:       7000
TRAIN LOSS:  0.002697316797355325
TEST LOSS:   0.0010223792958401138
```

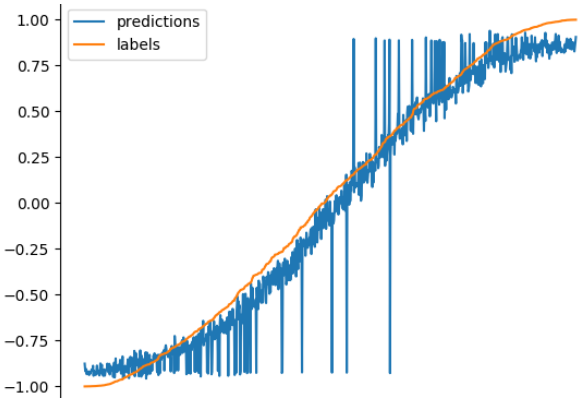
```
EPOCH:       8000
TRAIN LOSS:  0.0032573881926539774
TEST LOSS:   0.007223008289809672

EPOCH:       9000
TRAIN LOSS:  0.002555384577707217
TEST LOSS:   0.0010074300561906569

EPOCH:       9999
TRAIN LOSS:  0.0025873418696507285
TEST LOSS:   0.002436269773668111
```

It is dificult to make an interpretation of how good/bad these values are, as they are relative to the values of the MLP output and the labels of the data set. To give a better visualisation of how sufficiently well the MLP learned, after training I then generated a new random data set of 50 examples again, of the same format as previously. I sorted these examples by their label values, made a prediction for each input vector, and plotted the labels vs the predictions, as shown here:

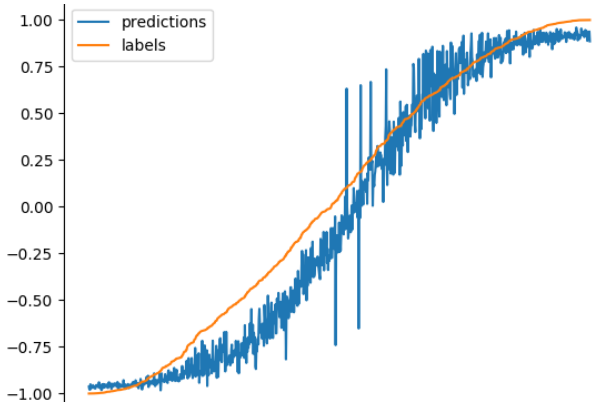


Both of these plots were produced after training with all parameters identical. This shows how important the initial weights values of the MLP are when beginning training. Although the right plot is far better, both do seem to manage to learn the general trend of the sin function, but the first plot is clearly far more volatile in its exact values.

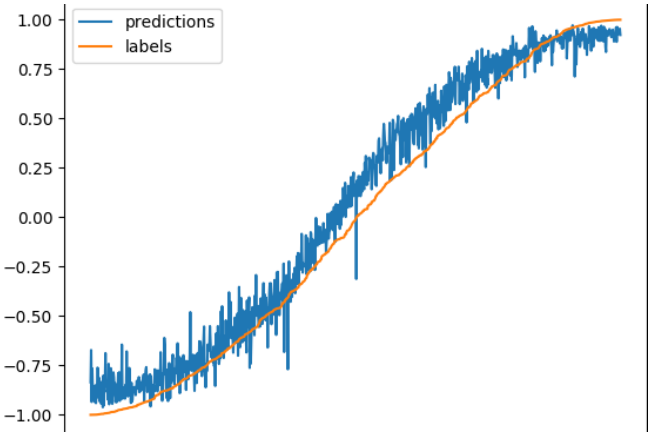
In attempt to improve on this, I decided to use a larger data set to avoid the possibility of overfitting, as well as increasing the number of hidden neurons to increase the predictive power of the MLP. I now generated the data set with 10,000 examples, as opposed to 50. The results for different number of hidden neurons and training epochs are shown below:



5 hidden neurons, 10,000 epochs



100 hidden neurons, 10,000 epochs



100 hidden neurons, 50,000 epochs

Evidently, there is a clear relationship that as the architecture size of the MLP increases, and it trains for longer (without overfitting occurring), the predicted trend fits more closely to the actual sin trend. With all parameters, the general, overall trend of the function seems to be learnt sufficiently well, but the error around this general trend still remains large and volatile in all cases.

I think that, given that the regression problem being learned here is essentially two functions (a summation with the sin function then applied to it), I believe that adding more layers to the MLP, thus giving the effect of stacking more non-linear activation functions on top of each other, would give the MLP far more flexibility when learning this problem.

Letter Recognition

This script can be run as

```
python LetterRecognition.py [load]
```

where if ‘load’ is not included, a new MLP will be initialised and trained on the data set, and if ‘load’ is included, an MLP that I have pre-trained and included will be loaded and its loss and accuracy on both the training and test sets will be determined.

This task was the main reason I chose to implement the Softmax activation function, as there are 26 output classes, all of which being mutually exclusive.

Through experimenting, I found that updating the weights throughout each epoch, rather than once after each epoch produced the best results. I arbitrarily chose to update the weights after every 2,000 examples were processed, thus updating them 10 times per epoch. As well as this, without normalising the input data, many problems occurred. As a result, I used Min Max Normalisation on the input features, which in this case was simply dividing each value by 15, thus fitting them all to the range [0, 1].

Through experimenting, I found that a learning rate of 0.0005 was sufficient for all hidden activation functions, and thus I ran a test to determine which hidden activation produced the best results, and to compare the difference in accuracy between a relatively small and large hidden layer. I ran a loop to train 1,000 epochs on MLPs with 10 and 100 hidden units, for each hidden activation of sigmoid, tanh, and ReLU.

```
SIGMOID 10
TRAIN LOSS:  0.9202531575996378
TRAIN ACC:   74.29 %
TEST LOSS:   0.9835192744762707
TEST ACC:    72.10 %

SIGMOID 100
TRAIN LOSS:  0.6195099794297375
TRAIN ACC:   82.33 %
TEST LOSS:   0.6726217678673162
TEST ACC:    81.65 %

TANH 10
TRAIN LOSS:  1.299714412800753
TRAIN ACC:   61.84 %
TEST LOSS:   1.3211631090501172
TEST ACC:    60.72 %

TANH 100
TRAIN LOSS:  0.2941544206086196
TRAIN ACC:   91.27 %
TEST LOSS:   0.3493040494200178
TEST ACC:    89.50 %
```

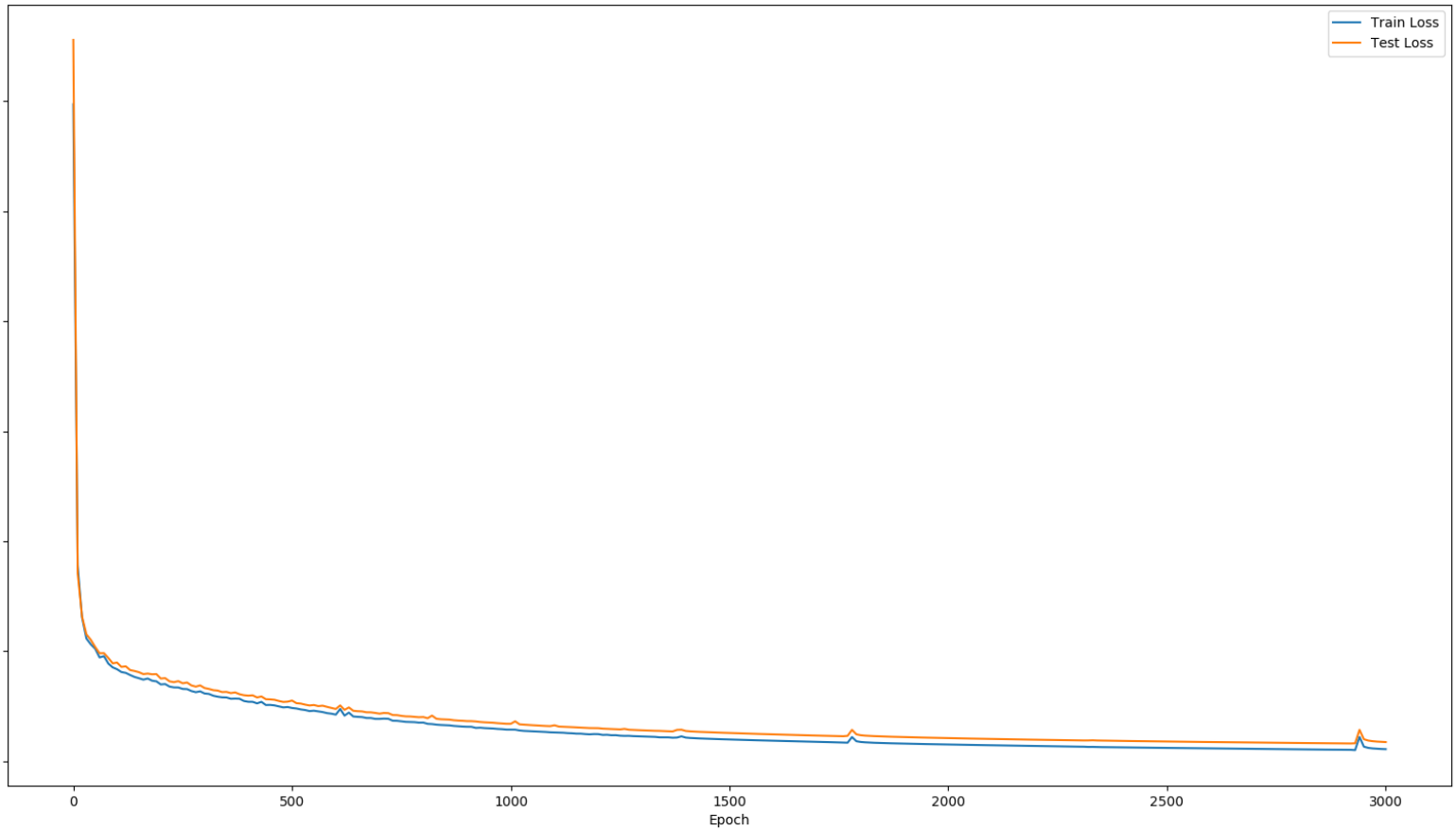
```
RELU 10
TRAIN LOSS:  0.8187549876846995
TRAIN ACC:   77.13 %
TEST LOSS:   0.8429721368541108
TEST ACC:    77.05 %

RELU 100
TRAIN LOSS:  0.7116147343482598
TRAIN ACC:   78.84 %
TEST LOSS:   0.8360834641994478
TEST ACC:    76.22 %
```

Very interestingly, the tanh MLP performed the worst out of any activation functions with 10 hidden neurons, but was by far the most superior with 100 hidden neurons. This was the MLP I chose to train for further epochs (3,000) to see what test accuracy could be achieved. I was impressed to see the following best results after this training session:

```
TRAIN LOSS:  0.10430746849205702
TRAIN ACC:   97.12 %
TEST LOSS:   0.16316118878900224
TEST ACC:    94.97 %
```

When the model is training, it repeatedly evaluates itself on the test set every *n* epochs, and if the loss achieved here is the lowest thus far, it stores all the parameters of the current state of the MLP. This is the same model that will be loaded if the script is ran with the 'load' option.

I plotted out the loss over this training session which is shown on the following page. I noticed that the loss was still decreasing steadily for both the training and test sets after these 3,000 epochs, so I think with even more training epochs, as well as a method for varying the learning rate appropriately, rather than leaving it constant, would be able to increase the accuracy further closer to 100%.


Train Loss
Test Loss
Loss
Epoch
0
1
2
3
4
5
6
0
500
1000
1500
2000
2500
3000